Chapter 7

Evaluating the Instruction Set Architecture of H1: Part 1

We will study the assembly code generated by a “dumb” C++ compiler

- Will provide a sense of what constitutes a good architecture.
- Will provide a better understanding of high-level languages, such as C++ and Java.

Dumb compiler

- Translates each statement in isolation of the statements that precede and follow it.
- Dumb compiler lacks all non-essential capabilities, except for a few that would *minimally* impact its complexity.

Dump compiler translates

```
x = 2;
y = x;     to

 ldc  2
 st   x

 ld   x        ; this ld is unnecessary
 st   y
```

Compiler does not remember what it did for the first instruction when it translates the second.

A smart compiler would translate

```
x = 2;
y = x;        to
```

```
ldc  2
st   x


st   y
```

Smart compiler is able to generate more efficient code for the second instruction by remembering what it did for the first instruction.

Dumb compiler generates code left to right

```
d = a + b + c;
```

where **a**, **b**, **c**, and **d** are global variables, is translated to

```
ld    a            ; access a first
add   b            ; access b next
add   c            ; access c last
st    d
```

Dumb compiler follows order required by operator precedence and associativity.

* has higher precedence than +
= is right associative.

```
d = a + b*c;    // code for b* c first
a = b = c = 5; // code for c = 5 first
```

When dumb compiler generates code for a binary operator, it accesses operands in the order that yields better code.

```
a = b + 1;
```

our compiler can generate an `ldc` instruction to get the constant 1, but only if it generates this code before it generates code to access **b**:

```
ldc   1
add   b
st    a
```

If **b** were accessed first, then the `ldc` instruction could not be used to load 1 (it would destroy the value of **b** in the **ac** register). Thus, the compiler would have to generate

```
ld    b
add   @1
st    a
```

where **@1** is a compiler-generated label defined as

```
@1:   dw    1
```

Dumb compiler always follows order imposed by parentheses

$$v = (w \underset{\substack{\uparrow \\ 1^{st}}}{+} x) \underset{\substack{\uparrow \\ 3^{rd}}}{+} (\, y \underset{\substack{\uparrow \\ 2^{nd}}}{+} z);$$

Other orders might yield more efficient code.

Dumb compiler performs constant folding (compile-time evaluation of constant expressions)

```
x = 2 + 3;
```

a compiler can generate

```
ldc  2
add  @3
st   x
```

Run-time evaluation

where **@3** is defined as

```
@3:  dw    3
```

Thus, at run time the `add` instruction computes the value of 2 + 3. Alternatively, a compiler can add 2 and 3 at compile time to get 5, and then generate code to load and store 5:

```
ldc  5
st   x
```

Compile-time evaluation

Constant folding only on constants at the *beginning* of an expression.

```
y = 1 + 2 + x;     // constant folding
y = 1 + x + 2;     // no constant folding
y = x + 1 + 2;     // no constant folding
```

Global variables in C++

- Declared outside of a function definition.
- Scope is from point of declaration to end of file (excluding functions with identically named local variable).
- Default to an initial value of 0 if an initial value is not explicitly specified.
- Are translated to a dw statement in assembler code.

FIGURE 7.1

```
#include <iostream>
using namespace std;

int x;              // global variable with no initial value specified
void fa()
{                   // cannot reference y from within fa
    x = x + x + 2;
}
int y = 3;          // global variable with initial value specified
void fb()
{
    x = 5;
    y = -2;
    fa();
}
void main()
{
    fb();
    cout << "x = " << x << endl;        // displays "x = 12"
    cout << "y = " << y << endl;        // displays "y = -2"
}
```

x and y in preceding slide are translated to

```
x:      dw      0       ; 0 default value
y:      dw      3
```

Direct instructions are used to access global variables

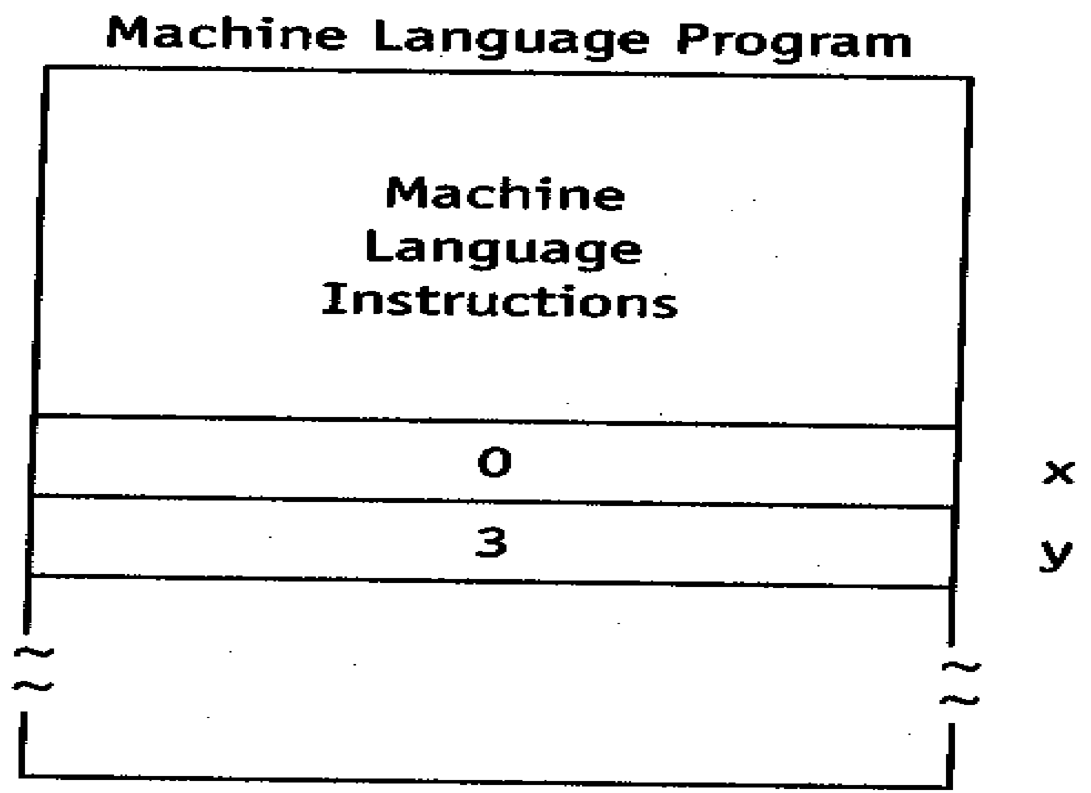
x = 5;

is translated to

```
ldc     5
st      x      ; use direct instruction
```

Global variables are part of the machine language program

FIGURE 7.2



Compiler generated names start with ‘@’

@2 is the label generated for the constant 2
@_2 is the label generated for the constant -2

@m0, @m1, ... , are for string constants

See @2, @_2, @m0, @m1 in the next slide.

FIGURE 7.3

```
fa:          ld       x              ; x = x + x + 2;
             add      x
             add      @2
             st       x

             ret
;==================================================================
fb:          ldc      5              ; x = 5;
             st       x

             ld       @_2            ; y = -2;
             st       y

             call     fa             ; fa()

             ret
;==================================================================
main:        call     fb             ; fb()

             ldc      @m0            ; cout << "x = " << x  << endl;
             sout
             ld       x
             dout
             ldc      '\n'
             aout

             ldc      @m1            ; cout << "y = " << y  << endl;
             sout
             ld       y
             dout
             ldc      '\n'
             aout

             halt
x:           dw       0              ; global variable
y:           dw       3              ; global variable
@2:          dw       2              ; @2 is a compiler-generated name
@_2:         dw       -2             ; @_2 is a compiler-generated name
@m0:         dw       "x = "         ; @m0 is a compiler-generated name
@m1:         dw       "y = "         ; @m1 is a compiler-generated name
             end main
```

A *local variable* can be referenced by name only in the function or sub-block within the function in which it is defined.

- Two types of local variables: *dynamic* and *static*.
- Static local variables are defined with a dw statement. Direct instructions are used to access them. Created at *assembly time*.
- Dynamic local variables are created on the stack at *run time*. Relative instructions are used to access them.

FIGURE 7.4

```
#include <iostream>
using namespace std;

void fc()
{
      int dla ;                         // dynamic local variable
      int dlb = 7;                      // dynamic local variable

      static int sla;                   // static local variable
      static int slb = 5;               // static local variable

      cout << sla << endl;
      cout << slb << endl;
      dla = 25;
      dlb = 26;
      sla = 27;
      slb = 28;

}
void main()
{
      fc();
      // sla and slb retain values 27 and 28 between calls
      fc();
}
```

Output from preceding program

0	(initial value of **sla** on first call of **fc**)
5	(initial value of **slb** on first call of **fc**)
27	(initial value of **sla** on second call of **fc**)
28	(initial value of **slb** on second call of **fc**)

sla, slb translated to

```
@s0_sla:      dw      0
@s1_slb:      dw      5
```

Thus, the C++ statements

```
sla = 27;
slb = 28;
```

are translated to

```
ldc   27
st    @s0_sla
ldc   28
st    @s1_slb
```

Why are the names of static local variables not carried over *as is* (as with global variables) to assembler code?

See the next slide.

FIGURE 7.5

```
        C++ Program                          Assembler Program

        int x;
        void f1()                            f1:
        {                                            .
              static int x;                          .
              .                                      .
              .
              .                                      ret
        }
        void f2()                            f2:
        {                                            .
              static int x;                          .
              .                                      .
              .
              .                                      ret
        }
        void main()                          main:
        {                                            .
              .                                      .
              .                                      .
              .                                      halt
        }                                            .
                                                     .
                                             x:      dw     0    ; global
                                             x:      dw     0    ; static local in f1
                                             x:      dw     0    ; static local in f2

                                         Error: Duplicate labels
```

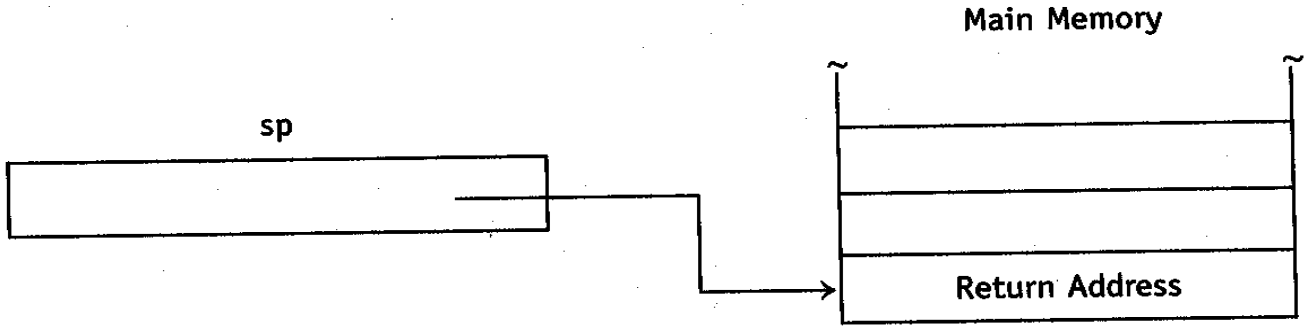
A dynamic local variable is allocated on the stack with a push or aloc instruction.

push is used if the variable has to be initialized. aloc is used otherwise.

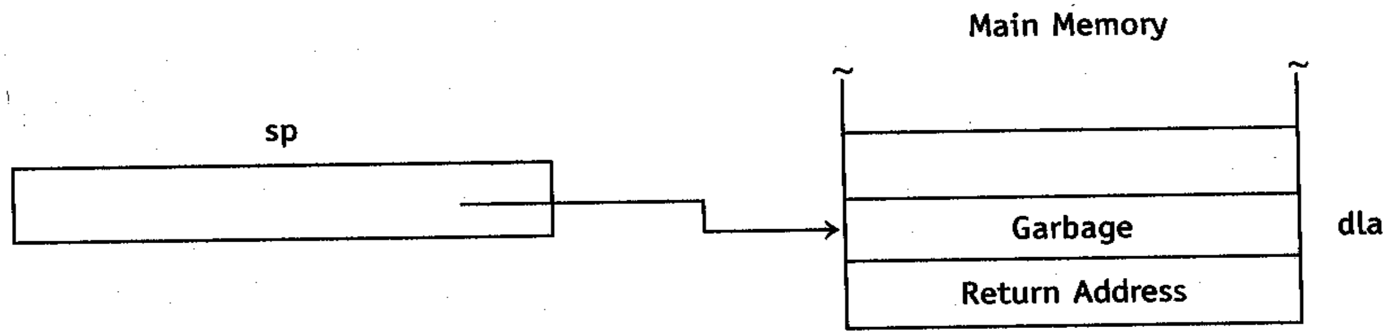
FIGURE 7.7

```
fc:          aloc 1          ; int dla;

             ldc  7          ; int dlb = 7;
             push
                  .
                  .
                  .
             dloc 2          ; deallocate dla and dlb
             ret
;==========================================

main:        call fc         ; fc();

             call fc         ; fc();

             halt
@s0_sla:     dw    0         ; static local variable sla
@s1_slb:     dw    5         ; static local variable slb
             end   main
```

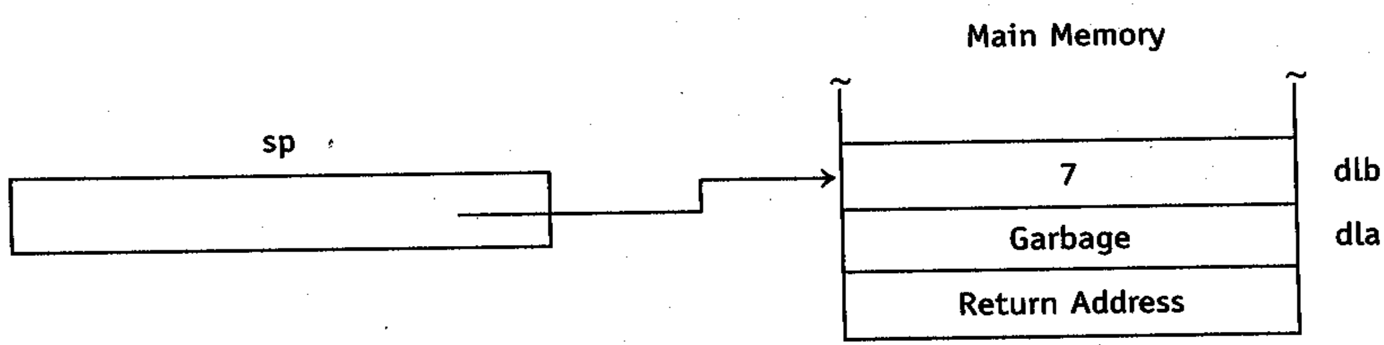
FIGURE 7.6 a) On entry into the function **fc**



b) After the creation of **d1a**



c) After the creation of **d1b**



The relative addresses of dla and dlb are 1 and 0, respectively.

See the preceding slide.

FIGURE 7.8

```
fc:         aloc 1          ; int dla;

            ldc 7           ; int dlb = 7;
            push

            ld @s0_sla      ; cout << sla << endl;
            dout
            ldc '\n'
            aout

            ld @s1_slb      ; cout << slb << endl:
            dout
            ldc '\n'
            aout

            ldc  25         ; dla = 25;
            str  1

            ldc  26         ; dlb = 26;
            str  0

            ldc  27         ; sla = 27;
            st   @s0_sla

            ldc  28         ; sla = 28;
            st   @s1_slb

            dloc 2          ; remove dla and dlb
            ret
;==============================================================
main:       call fc         ; fc();

            call fc         ; fc();

            halt
@s0_sla:    dw              0       ; static local variable sla
@s1_slb:    dw              5       ; static local variable slb
            end  main
```

The default initialization of global and static local variables (to 0) does not require additional time or space. But the default initialization of dynamic local variables would require extra time and space (a ld or ldc followed by a push versus a single aloc).

x and y initially 0; z initially has garbage

FIGURE 7.9

```
int x;                 // global, initial value = 0
void fd()
{
    static int y;      // static local, initial value = 0
    int z;             // dynamic local, initial value undefined
        .
        .
        .
}
void main()
{
    fd();
}
```

Default initializations are often not used, in which case the extra time and space required for the initialization of dynamic local variables would be wasted.

FIGURE 7.10

```
void fe() {
    int z;
      .
      .
    z = x + y;
      .
      .
}
```

Relative addresses can change during the execution of a function

- Changing relative addresses makes compiling a more difficult task—the compiler must keep track of the correct relative address.
- Changing relative addresses is an frequent source of bugs in hand-written assembly code for the H1.

Relative address of x changes

FIGURE 7.11

C++	Assembly Language
`void legal()`	`legal:`
`{`	
`    int x;`	`    aloc 1 ; allocate x`
`    x = 3;`	`    ldc  3`
	`    str  0  ; relative address of x is 0`
`    int y;`	`    aloc 1  ; allocate y--changes rel add of x`
`    x = 5;`	`    ldc  5`
	`    str  1  ; relative address of x is 1`
	`    dloc 2`
`}`	`    ret`

Call by value

- The value (as opposed to the address) of the argument is passed to a function.
- An argument and its corresponding parameter are distinct variables, occupying different memory locations.
- Parameters are created by the *calling function* on function call; destroyed by the *calling function* on function return.

m is the argument; x is its parameter; y is a local variable

FIGURE 7.12

```
int m = 5;                  // m is a global variable
void fg(int x)              // x is the parameter
{
    int y;                  // y is a dynamic local variable
    y = 7;
    x = y + 2;
}
void main()
{
    fg(m);                  // m is the argument
}
```

Creating parameters

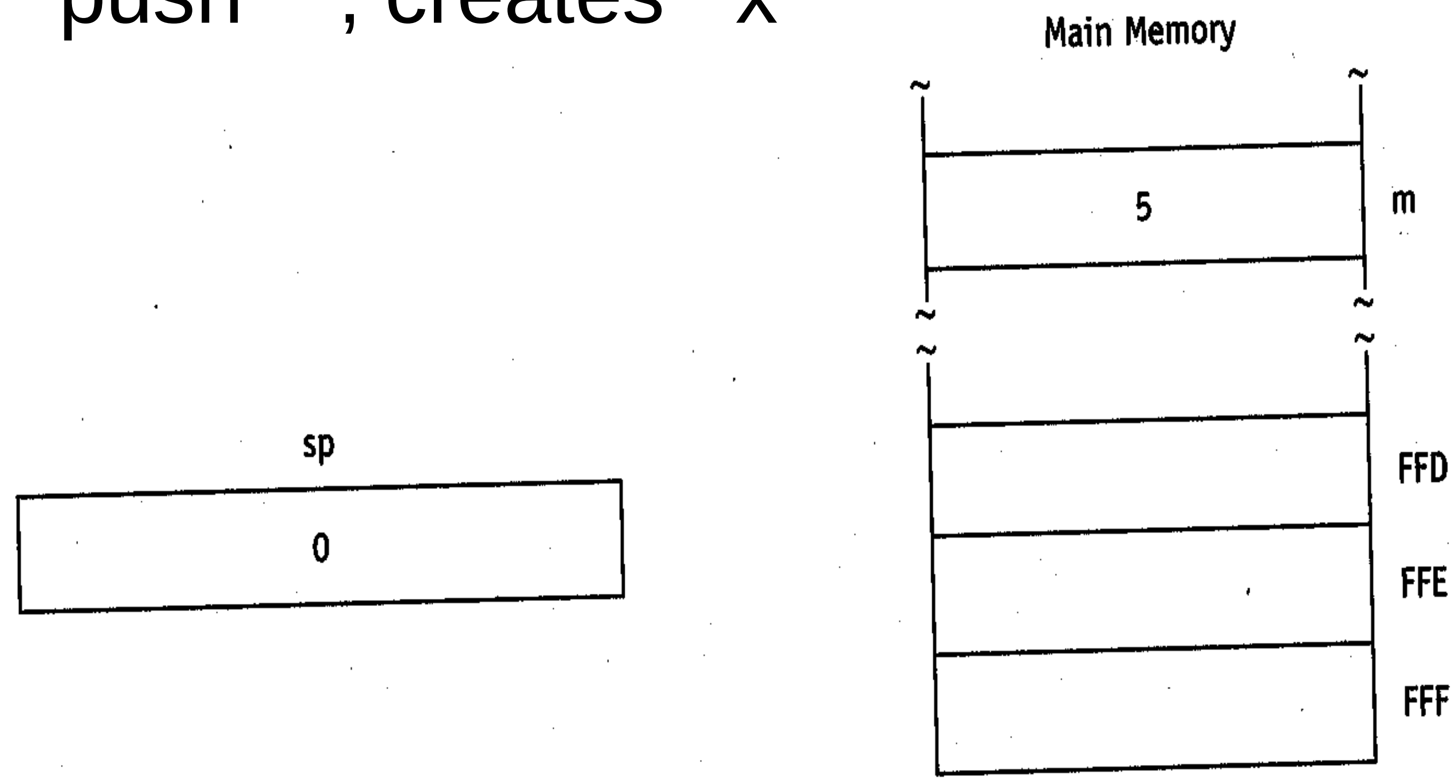
The parameter x comes into existence during the call of fg. x is created on the stack by pushing the value of its corresponding argument (m).

Start of function call f(m)

Push value or argument, thereby creating the parameter.

```
ld    m
push    ; creates  x
```

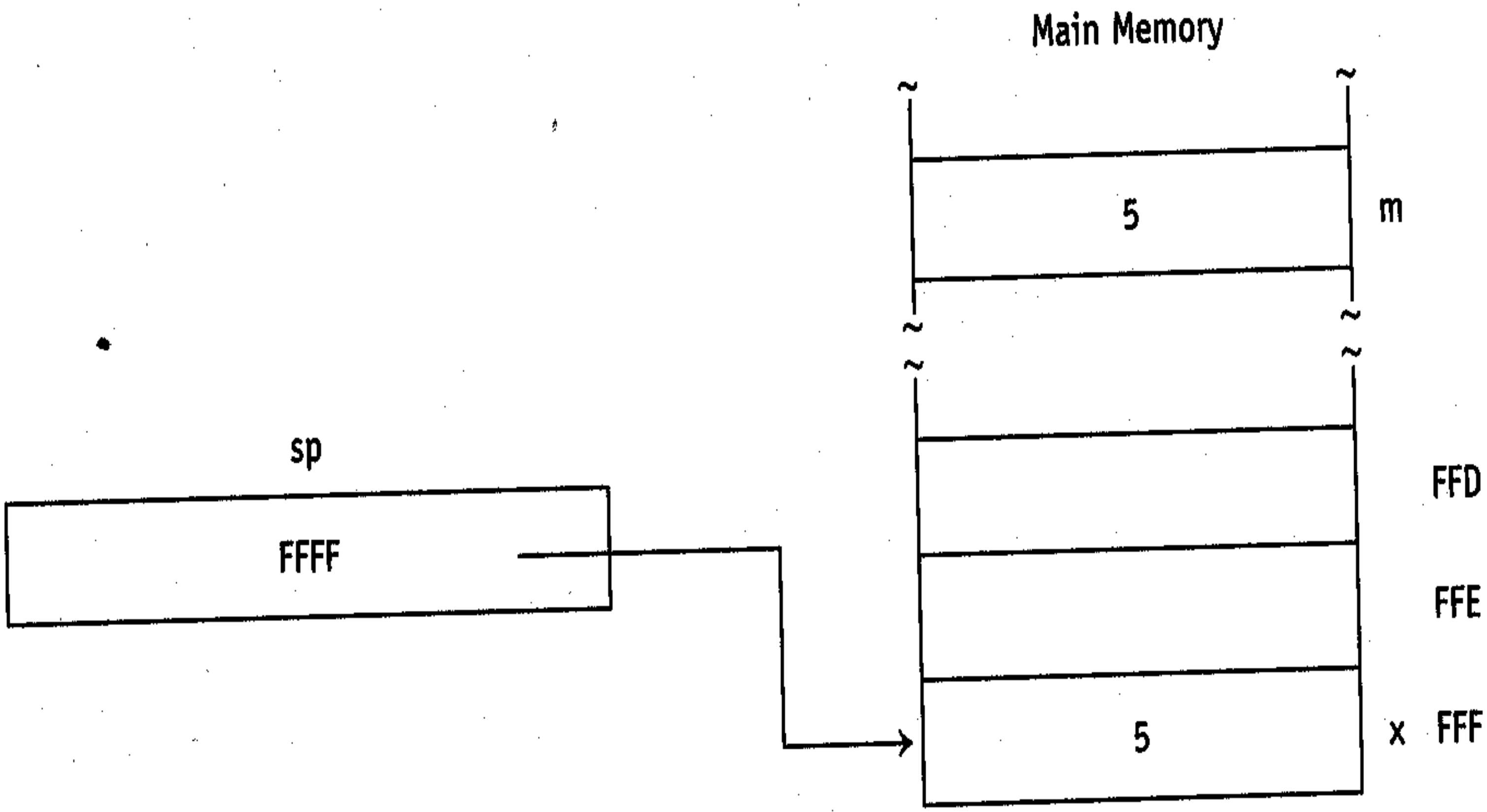
FIGURE 7.12A



Call the function (which pushes return address on the stack)

call fg

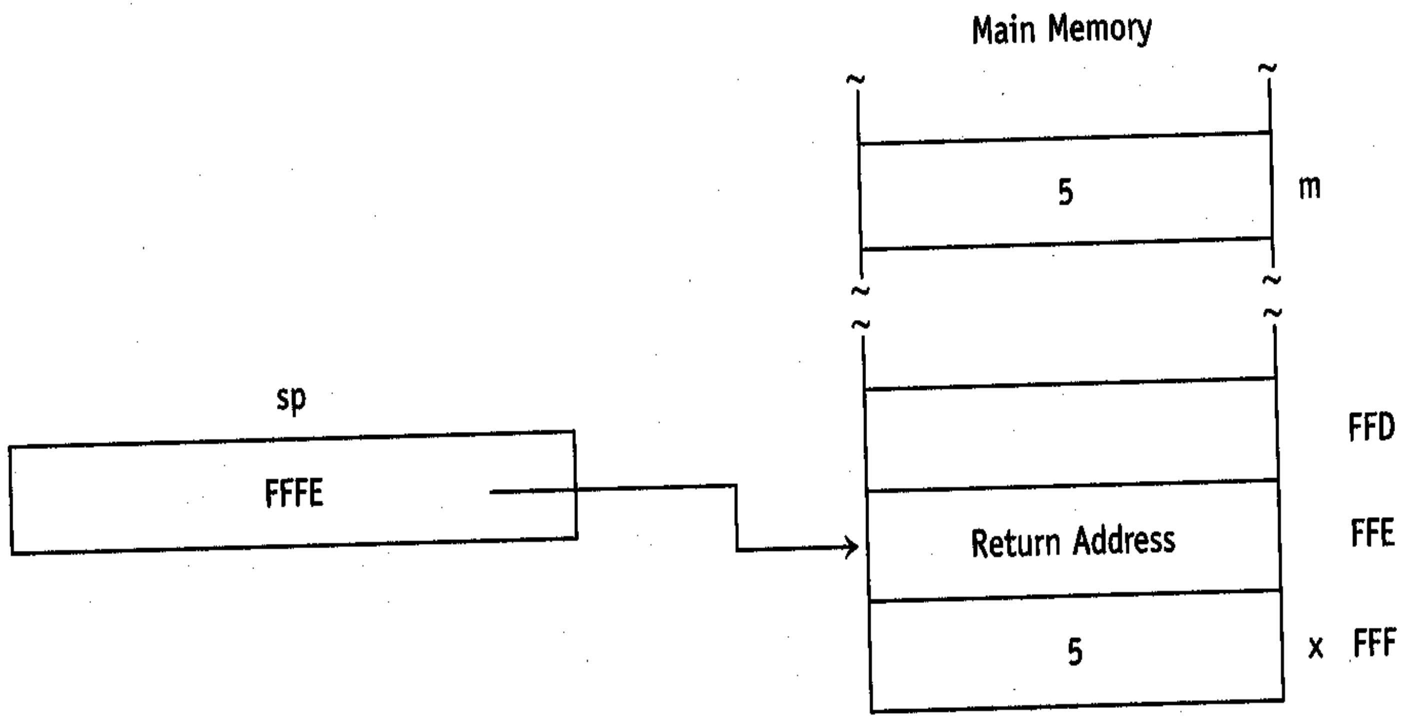
FIGURE 7.12B



Allocate uninitialized local variable with aloc

aloc 1

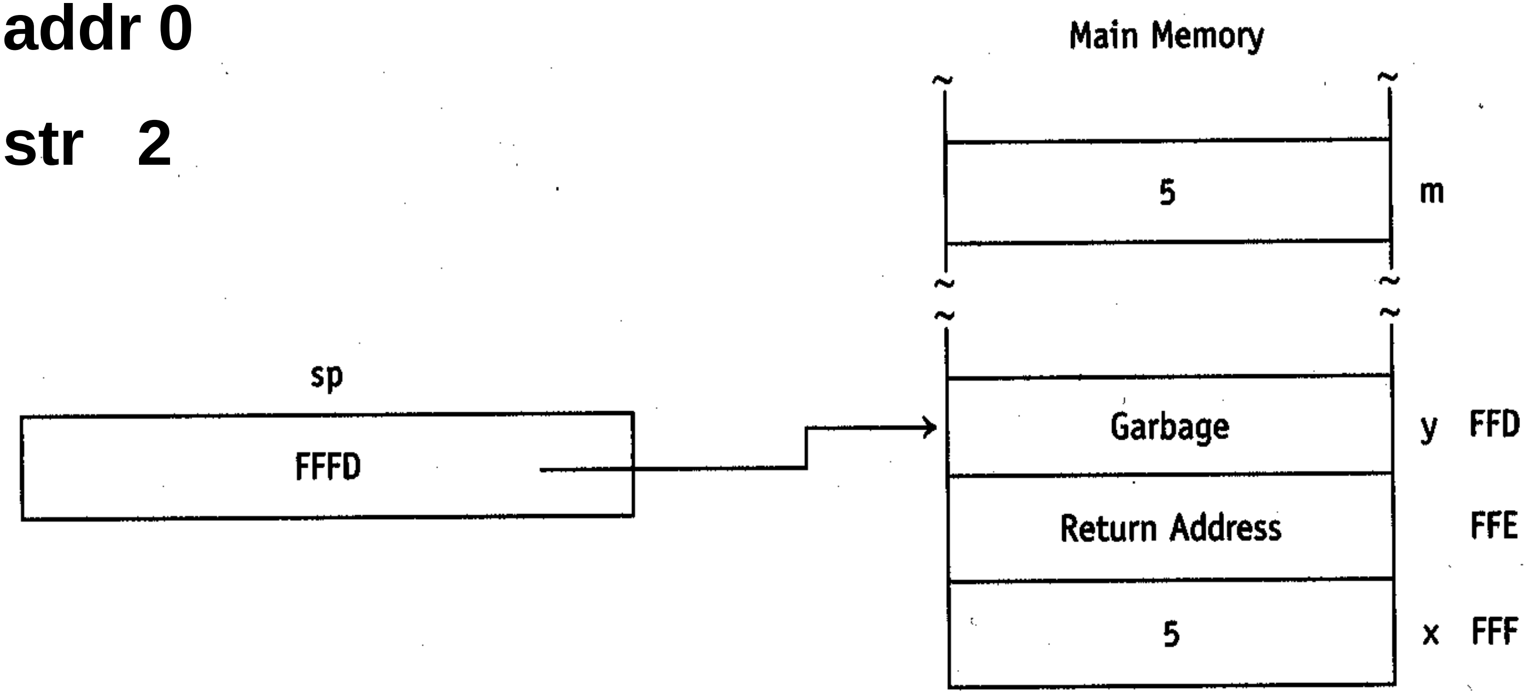
FIGURE 7.12C



Execute function body

```
ldc  7      ; y = 7;
str  0
ldc  2      ; x = y + 2;
addr 0
str  2
```

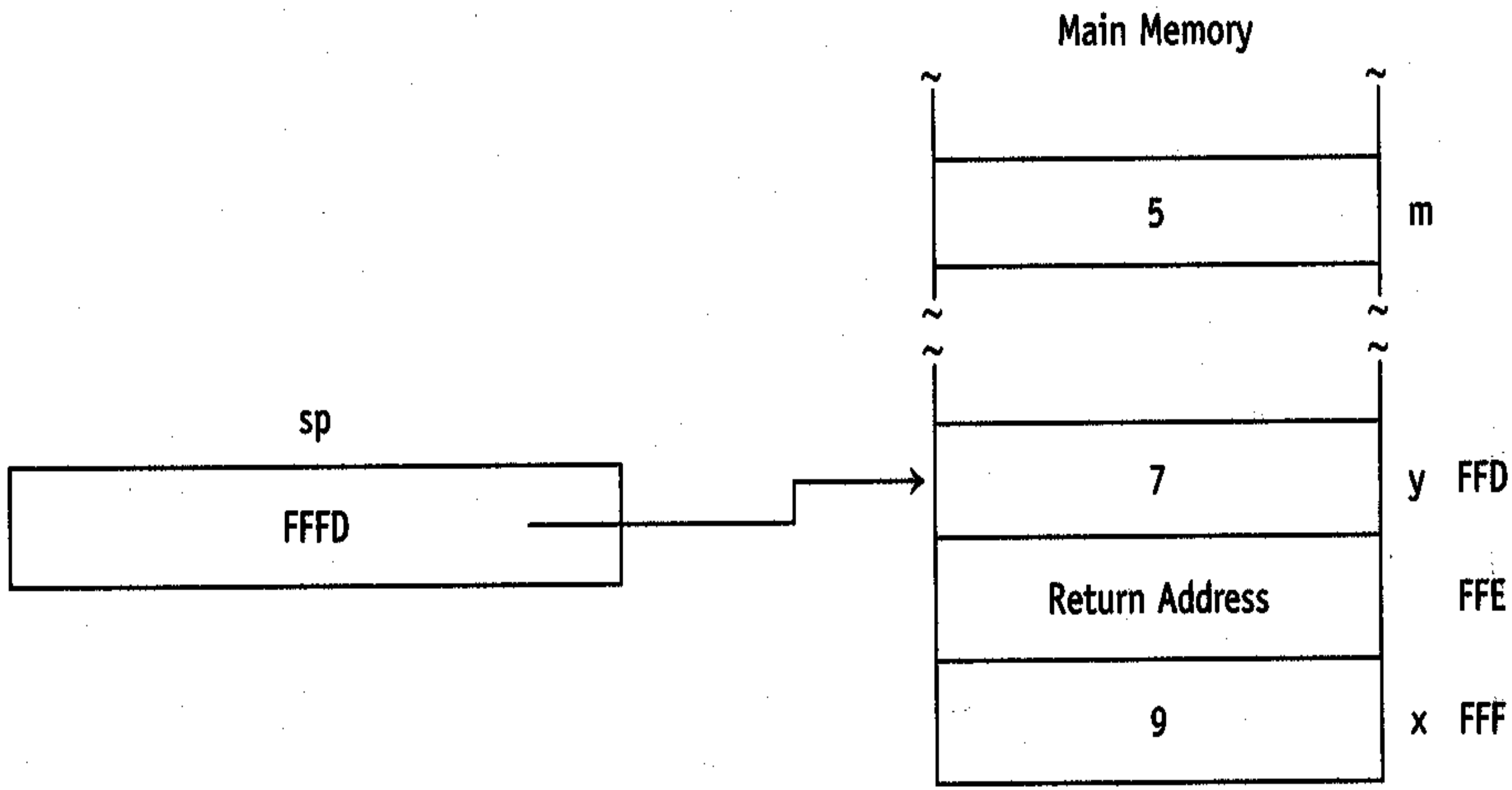
FIGURE 7.12D



Deallocate dynamic local variable

dloc 1

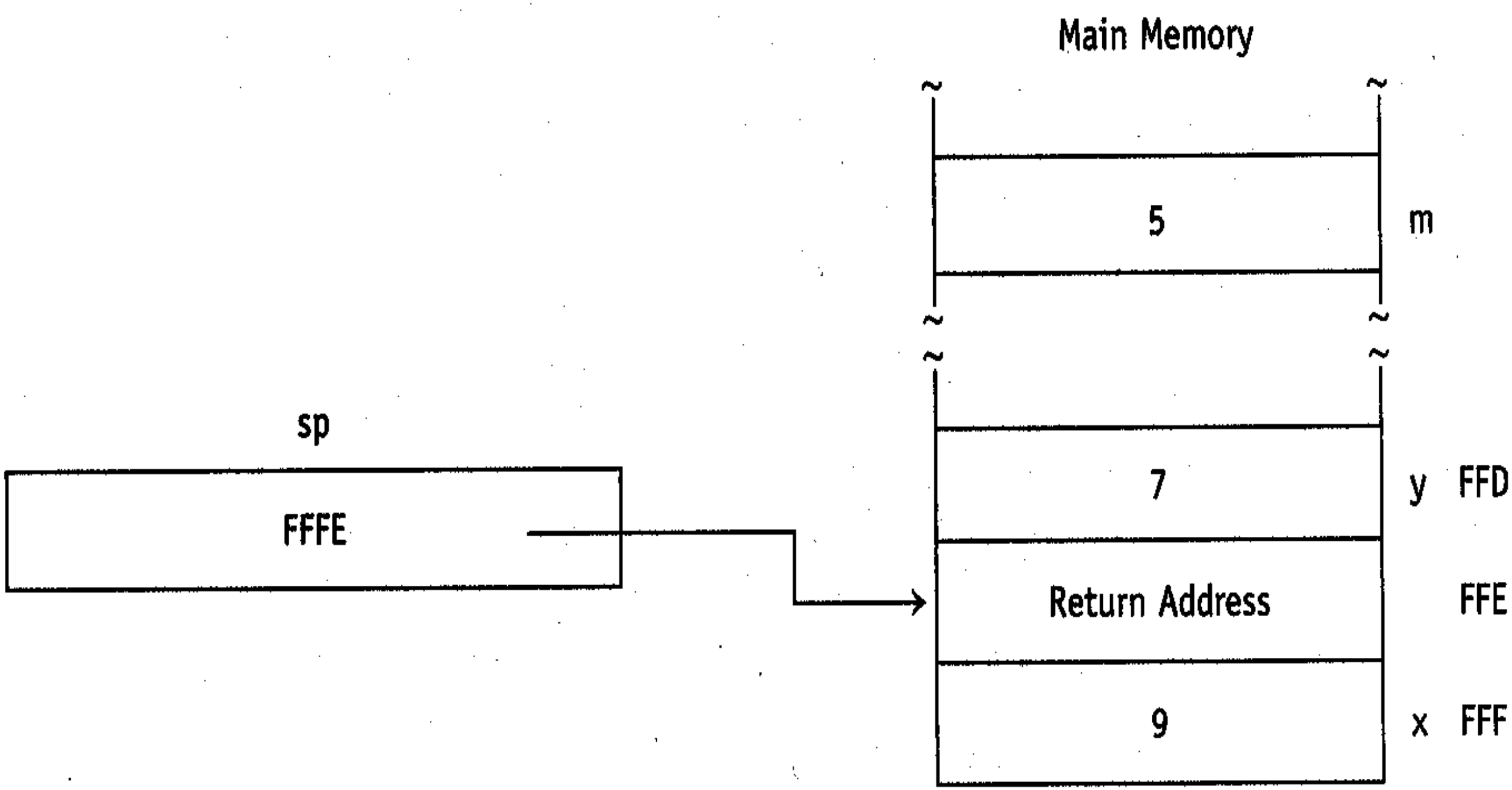
FIGURE 7.12E



Return to calling function (which pops the return address)

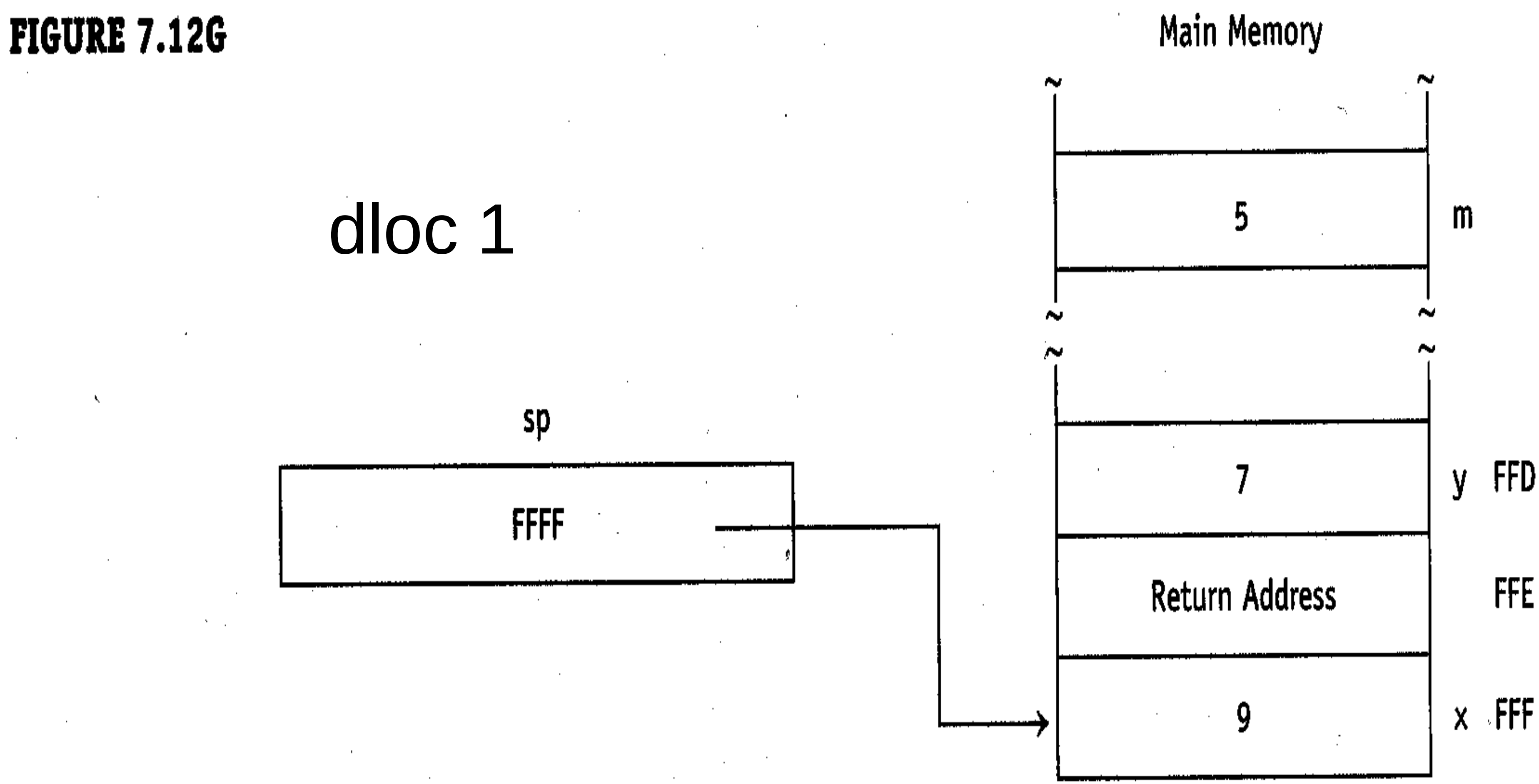
ret

FIGURE 7.12F



Deallocate parameter

dloc 1



Back to initial configuration

FIGURE 7.12H

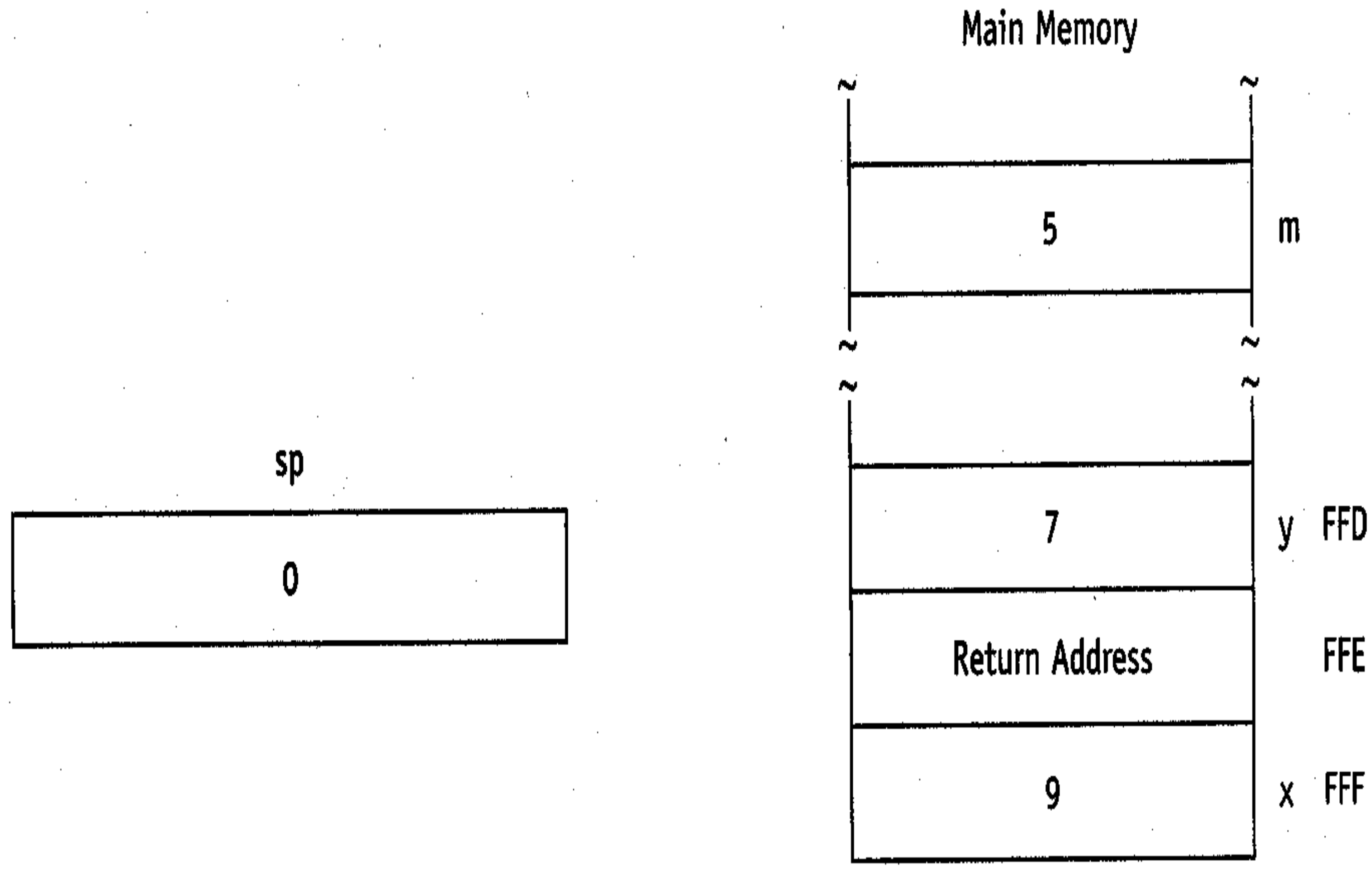


FIGURE 7.13

```
fg:         aloc 1            ; int y;
            ldc 7             ; y = 7;
            str 0
            ldc 2             ; x = y + 2;
            addr 0
            str 2
            dloc 1            ; deallocate y
            ret
;==============================================
main:       ld m              ; fg(m);
            push
            call fg
            dloc 1
            halt
m:          dw 5
            end  main
```

Parameter creation/destruction

The *calling function* creates and destroys parameters.

```
ld      m
push        ; create parameter x
call    fg
dloc    1   ; destroy parameter x
```

Dynamic local variable creation/destruction

The *called function* creates and destroys dynamic local variables.

```
aloc  1   ; create y.
.
.
dloc 1  ; destroy y
```

Push arguments right to left

```
void fh(int x, int y, int z)
{
   ...
}
```

If a, b, and c are global variables, then the call,

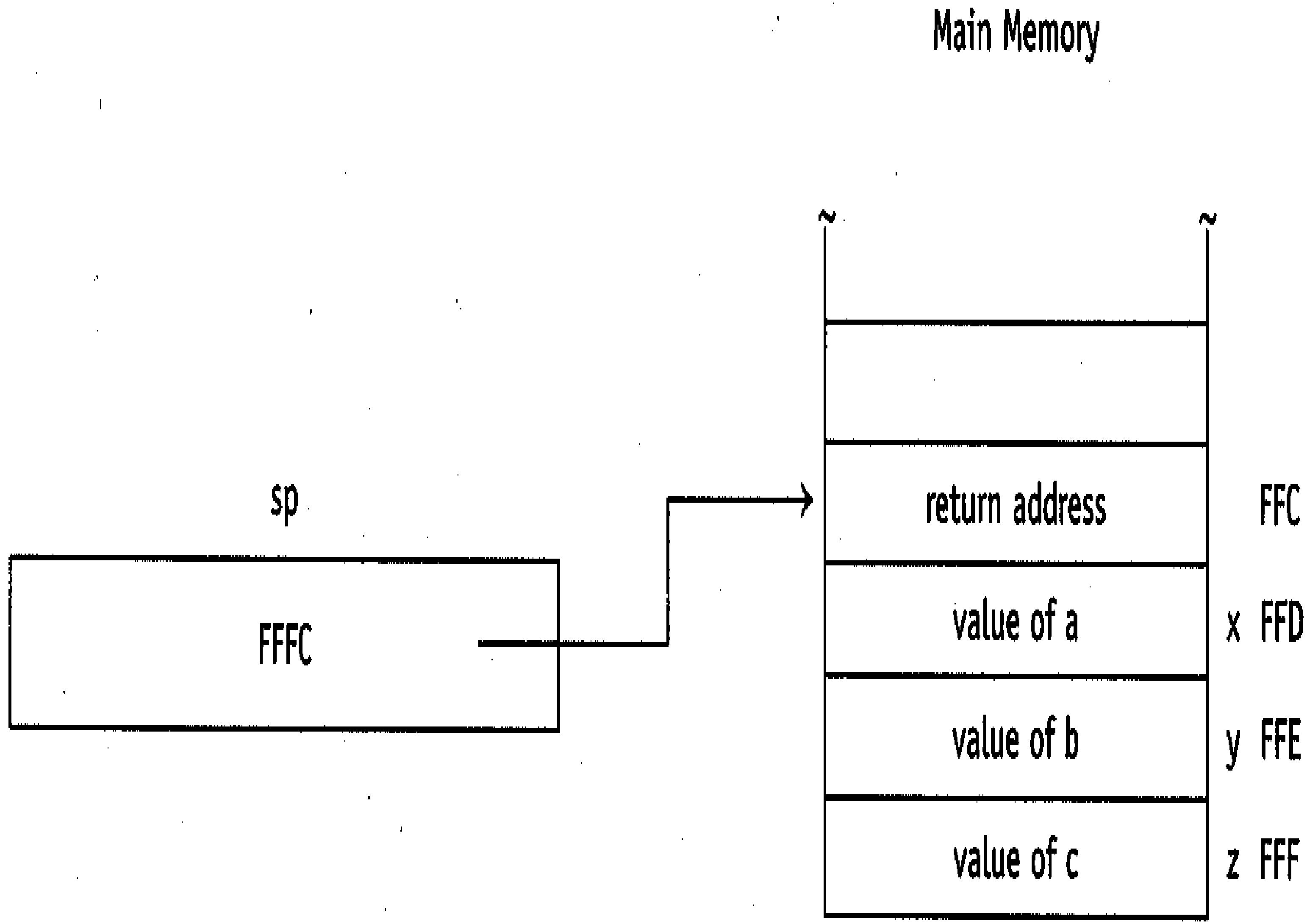
```
fh(a, b, c);
```

is translated to

```
ld    c
push        ; create z
ld    b
push        ; create y
ld    a
push        ; create x
call  fh
dloc  3     ; remove all three parameters
```

On entry into fh
x, y, and z have relative addresses 1, 2, and 3

FIGURE 7.14



The C++ return statement returns values via the ac register.

FIGURE 7.15

```
#include <iostream>
using namespace std;

int x, y;
int add_one(int z)
{
    return z + 1;
}
void main()
{
    x = 1;
    y = add_one(x);     // value returned is assigned to y
    cout << "y = " << y << endl;
}
```

Using the value returned in the ac

```
y = add_one(x);
```

is

```
ld   x          ; get value of x
push            ; create parameter on the stack
call add_one
dloc 1          ; remove parameter
st   y          ; use value returned in ac
```

FIGURE 7.16

```
add_one:    ldc   1          ;return z + 1;
            addr 1
            ret
;===========================================================
main:       ldc   1          ; x = 1;
            st    x
            ld    x          ; y = add_one(x);
            push
            call add_one
            dloc 1
            st    y
            ldc   @m0        ; cout << "y = " << y << endl;
            sout
            ld    y
            dout
            ldc   '\n'
            aout
            halt
x:          dw    0
y:          dw    0
@m0:        dw    "y = "
            end   main
```

Is it possible to replace relative instructions with direct instructions?

Answer: sometimes

It *is* possible to replace relative instructions accessing x, y, z with direct instructions in this program.

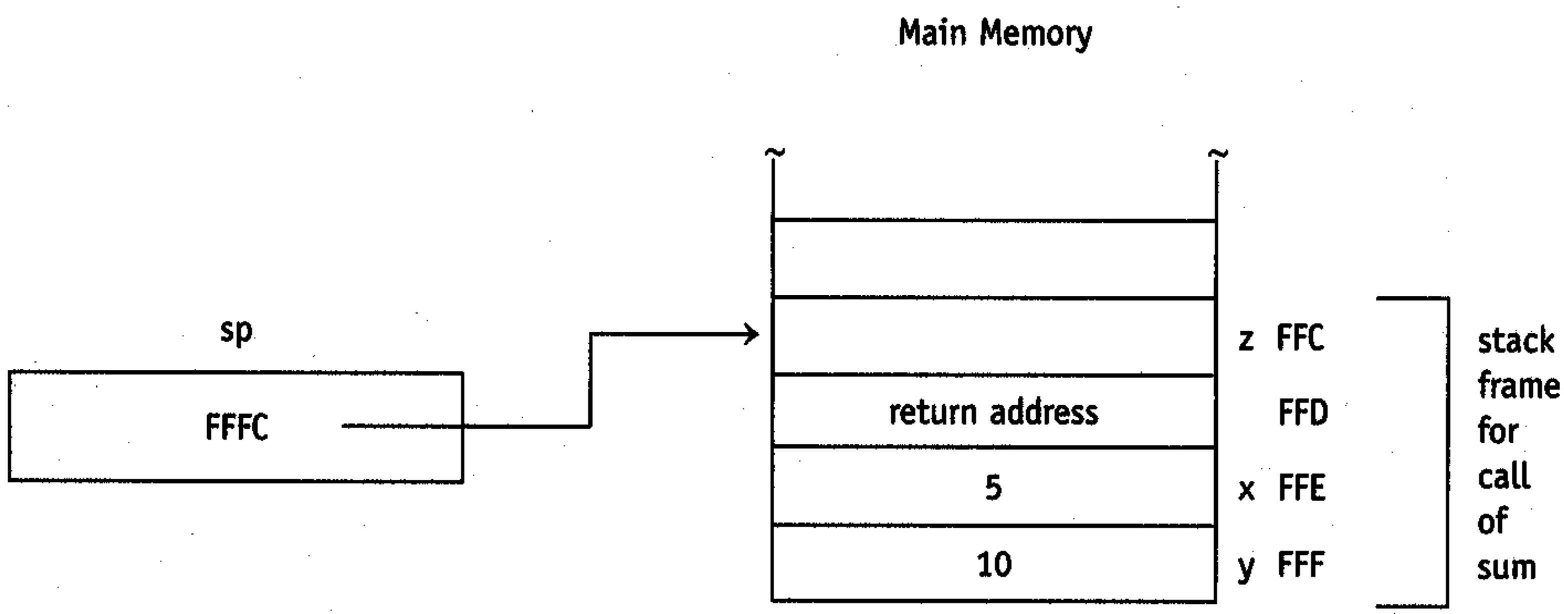
FIGURE 7.17

```
void sum(int x, int y)
{
    int z;

    z = x + y;
}
void main()
{
    sum(5, 10);
}
```

Can use either an absolute address or its corresponding relative address

	relative address	absolute address
x	2	FFE
y	3	FFF
z	0	FFC

FIGURE 7.18



Stack frame: the collection of parameters, return address and local variables allocated on the stack during a function call and execution.

A stack frame is sometimes called an *activation record*.

FIGURE 7.19

```
sum:     aloc 1
         ldr  2        ; replace with ld  FFEh
         addr 3        ; replace with add FFFh
         str  0        ; replace with st  FFCh
         dloc 1
         ret
;====================================================
main:    ldc  10       ; sum(5,10);
         push
         ldc  5
         push
         call sum
         dloc 2
         halt
         end  main
```

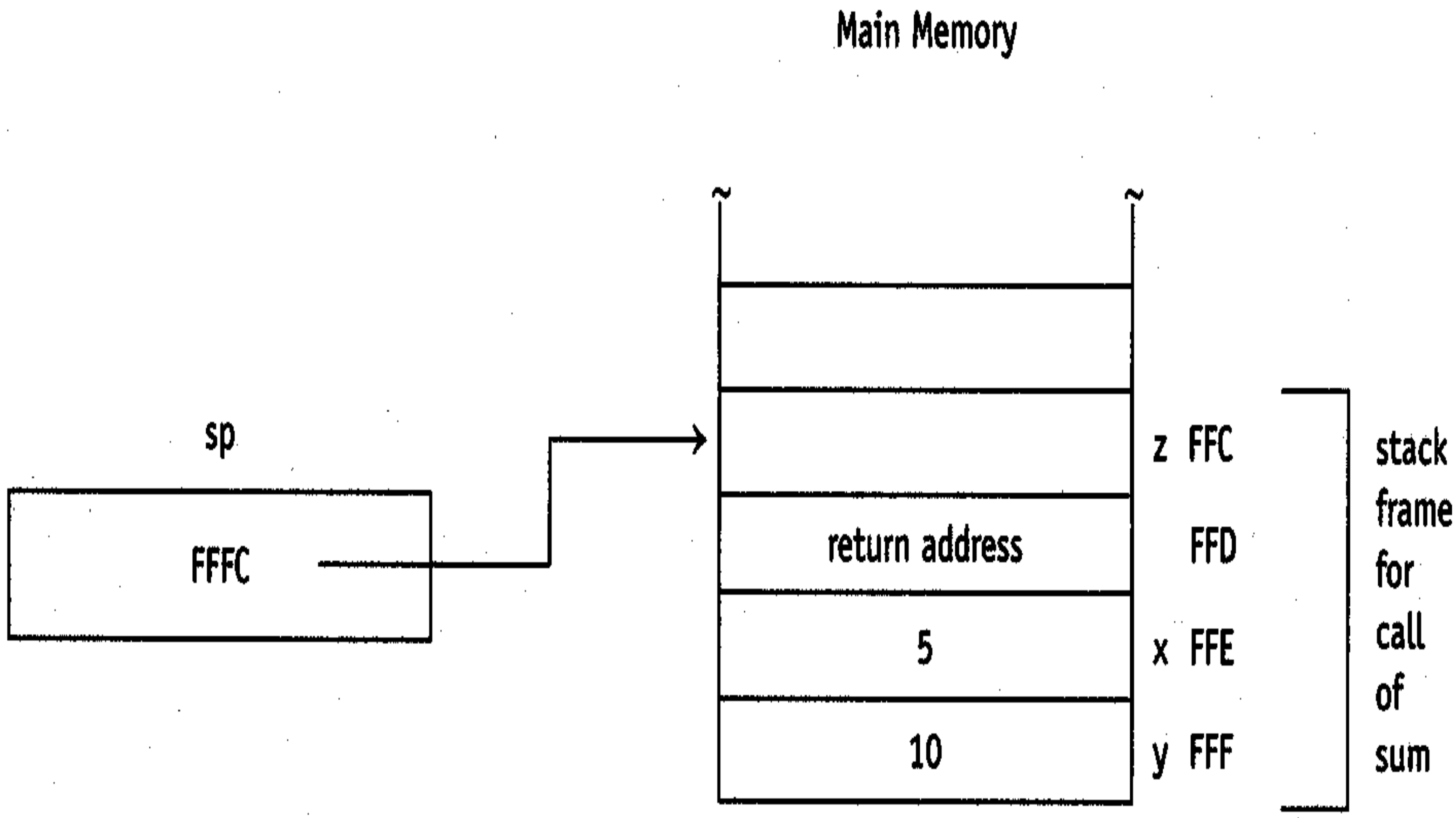
But for the following program, the *absolute addresses* of x, y, and z are different during the two calls of **sum**. But the *relative addresses* are the same.

FIGURE 7.20

```
void sum(int x, int y)
{
    int z;

    z = x + y;
}
void fk()
{
    sum(5, 10);
}
void main()
{
    sum(5, 10);
    fk();
}
```

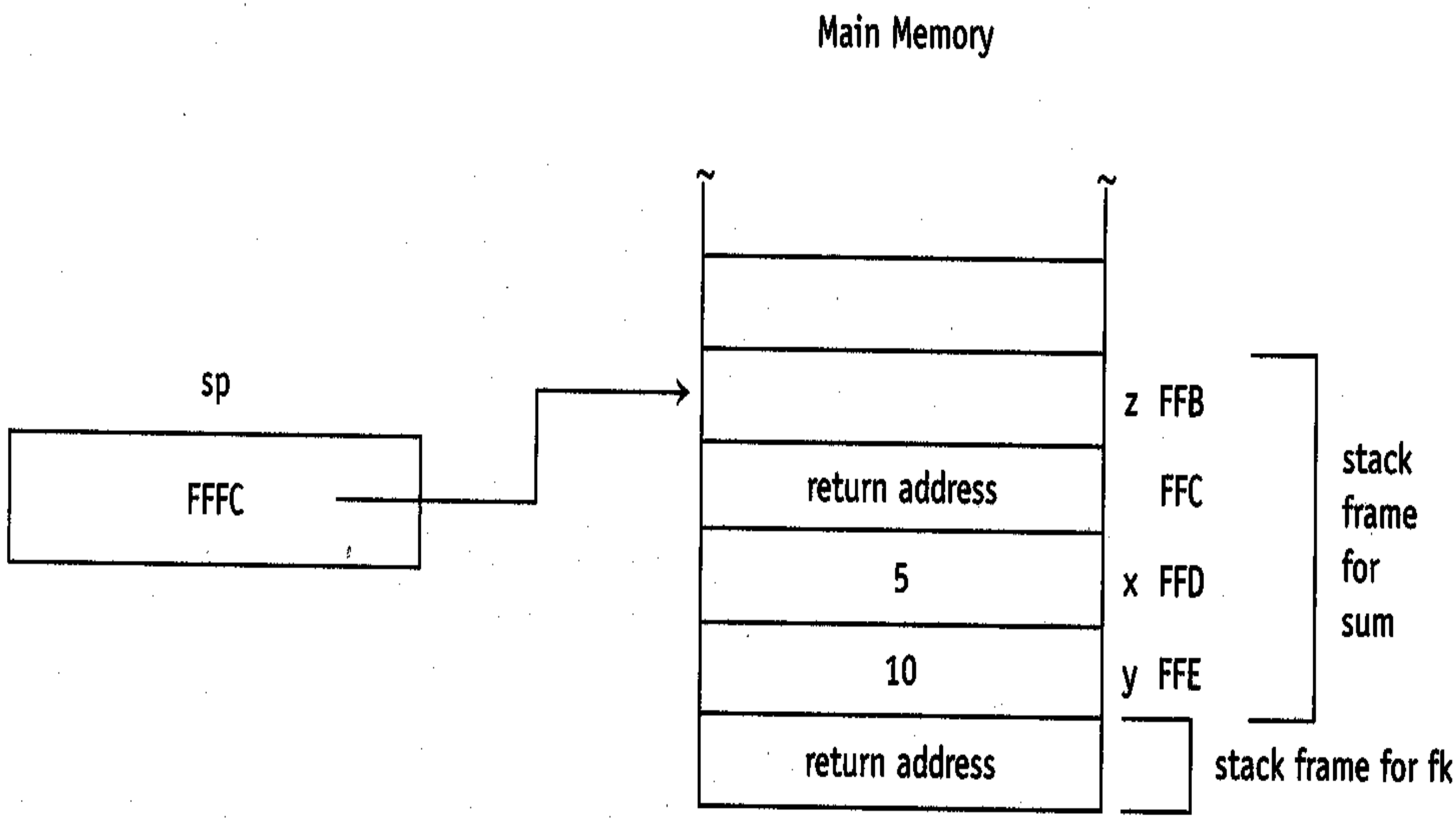
Stack frame for 1st call of sum

FIGURE 7.18



Stack frame for 2nd call of sum

FIGURE 7.21



Another problem with H1: determining the absolute address of items on the stack.

Determining the addresses of globals and static locals is easy, but difficult for dynamic locals.

Easy

Suppose **x** and **y** are global variables. Because **x** and **y** are defined with **dw** directives, the C++ statement

```
y = &x;      // "&" means "address of"
```

is translated to the assembly code

```
ldc  x    ; load address of x
st   y    ; store in y
```

Also easy

time. Now suppose **x** and **y** are static local variables. Then

```
y = &x;
```

would be translated to

```
ldc  @s0_x
st   @s1_y
```

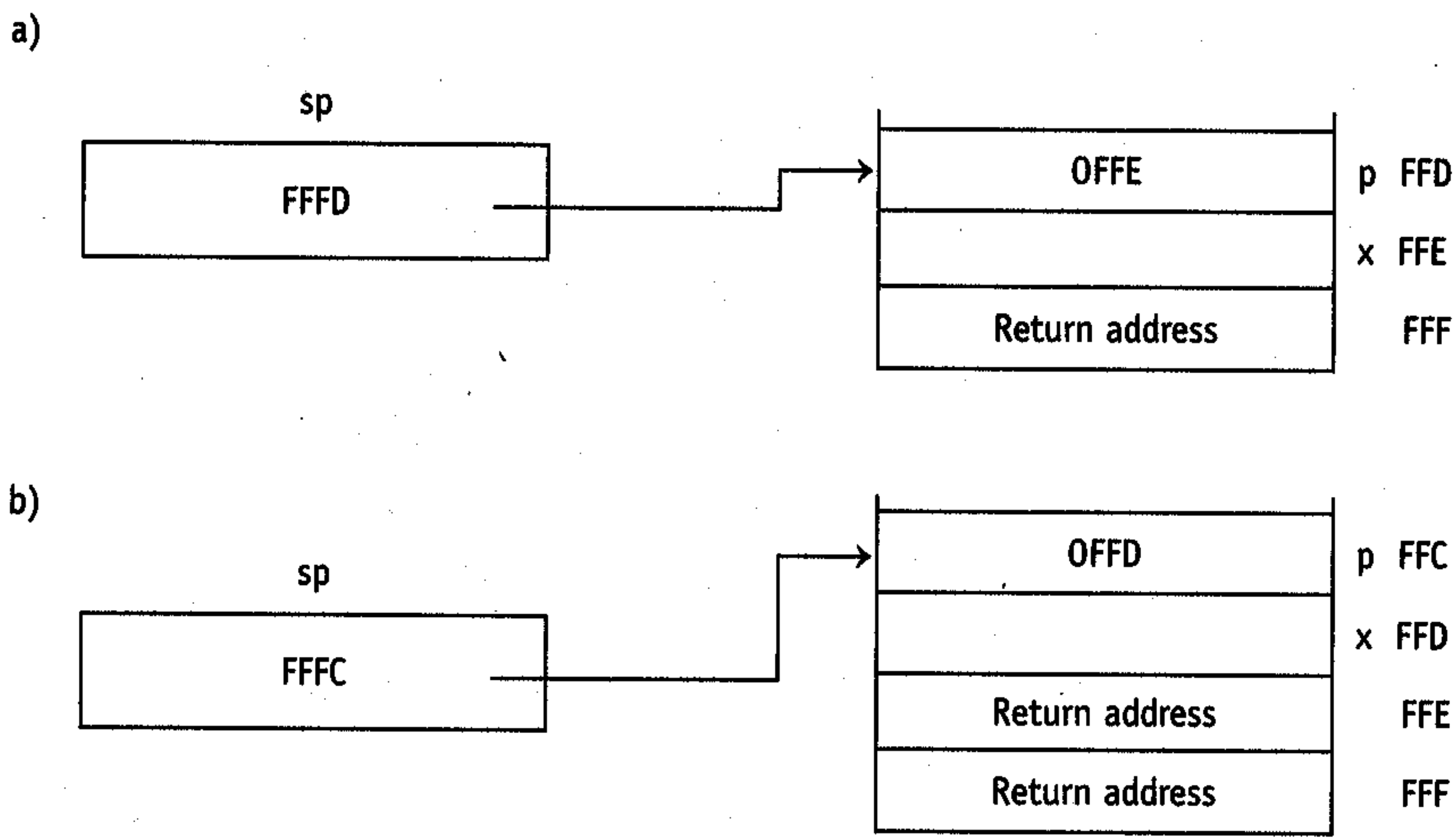
Relative address of x below is 1. What is its absolute address? It varies! Thus, &x must be computed at *run time* by converting its relative address to its corresponding absolute address.

FIGURE 7.22

```
void fm()
{
    int x;
    int *p;

    p = &x;          // assigning address of local variable
}
void fn()
{
    fm();
}
void main()
{
    fm();
    fn();
}
```

Absolute address of x different for 1st and 2nd calls of fm but relative address is the same (1).

FIGURE 7.23



To convert relative address 1 to absolute address:

```
        swap            ; corrupts sp
        st  @spsave
        swap            ; restores sp
        ldc  1
        add @spsave
```

where

```
@spsave:    dw     0
```

FIGURE 7.24

```
fm:          aloc 1            ; int x;
             aloc 1            ; int *p;
                               ; p = &x;
             swap                 ; get sp
             st   @spsave         ; save it
             swap                 ; restore sp
             ldc  1               ; get relative address of x (1)
             add  @spsave         ; convert to an absolute address
             str  0               ; store absolute address in p
             dloc 2               ; deallocate x and p
             ret
;            ======================================================
fn:          call fm            ; fm();
             ret
;            ======================================================
main:        call fm            ; fm();
             call fn            ; fn();
             halt
@spsave:     dw   0
             end  main
```

It is dangerous to corrupt the sp register on most computers, even for a short period of time.

An *interrupt mechanism* uses the stack. Interrupts can occur at any time. It the sp is in a corrupted state when an interrupt occurs, a system failure will result.

H1 does *not* have an interrupt mechanism.

Dereferencing pointers

Code generated depends on the type of the variable pointed to.

That is why you have to specify the type of the pointed-to item when declaring a pointer.

```
int    *p;       // p points to one-word number
long *q;         // q points to two-word number
```

```
*p = 0;
```

zeros one word, the `int` item pointed to by `p`. But

```
*q = 0;
```

zeros two words, the `long` item pointed to by `q`. For the former statement the compiler generates

```
ldc   0
push
ld    p         ; get address in p
sti
```

But for the latter statement the compiler has to generate a completely different sequence:

```
ldc   0         ; prepare for sti
push
ld    q         ; get address in q
sti             ; put zero in first word
ldc   0
push            ; prepare for sti
ldc   1         ; get 1
add   q         ; get (address in q) + 1
sti             ; put zero in second word
```